Journal of Computer and Knowledge Engineering, Vol. 8, No. 1, 2025. (1-10)

Ferdowsi University of Mashhad

Journal of Computer and Knowledge Engineering

https://cke.um.ac.ir

Information and Communication Technology Association of Iran

An Efficient Resource Allocation Algorithm for Task Offloading in the Internet of Vehicles*

Research Article

Ahmad Salehi[1], Sadoon Azizi[2]

10.22067/cke.2024.89721.1129

Abstract *The Internet of Vehicles (IoV) represents a transformative paradigm in Intelligent Transportation Systems (ITS), enabling real-time communication between vehicles, infrastructure, and cloud platforms to improve traffic management, safety, and efficiency. However, the resource limitations in vehicles pose significant challenges for delay-sensitive applications such as autonomous driving and automated navigation. Vehicular Edge Computing (VEC) offers a promising solution by offloading tasks to edge servers near vehicles, reducing transmission delays and enhancing computational efficiency. In this paper, we address the complex task offloading and resource allocation problem in VEC environments. We model this challenge as an Integer Linear Programming (ILP) problem, aiming to maximize the system's overall profit. To mitigate the computational complexity of solving the ILP problem, we propose an efficient heuristic algorithm. This approach considers various task types, accounting for the diversity and specific requirements of each. The algorithm optimizes CPU resource allocation based on task generation rates, average task sizes, and a calculated weight coefficient for each task type. Simulation results demonstrate that the proposed algorithm reduces memory costs and penalties from rejected tasks, while improving overall system profit. In particular, it outperforms existing algorithms by an average of 18.26% in terms of profit, demonstrating its effectiveness in practical VEC applications.*

Key Words Internet of vehicle, vehicular edge computing, task offloading, resource allocation, profit maximization.

1. Introduction

As modern automotive industries with sensing and wireless communication technologies rapidly advance, vehicles are becoming smarter, giving rise to the Internet of Vehicles (IoVs) as a new paradigm in Intelligent Transportation Systems (ITS) [1]. The IoVs combines vehicular ad hoc networks (VANETs) with the Internet of Things (IoT) to improve transportation efficiency and vehicle safety [2]. The rapid development of vehicular networks has enabled numerous delay-sensitive applications, including autonomous driving, automated navigation, vehicular augmented reality, and intelligent object recognition, each requiring substantial data processing and computational resources. These advancements are pushing current infrastructures to their limits, as they demand stringent Quality of Service (QoS) while processing large volumes of sensor data and communicating with the network [3]. However, the limited computational resources available within vehicles often prevent them from meeting the low-latency QoS requirements essential for these applications, creating a bottleneck in the advancement of vehicular networks.

To address these challenges, mobile edge computing (MEC) is considered a promising paradigm [4]. MEC enhances service efficiency for vehicles and reduces transmission delays between vehicles and cloud servers by bringing cloud computing resources closer to the network edge. Additionally, Vehicular Edge Computing (VEC) offers an effective solution to this problem [5]. In VEC, computational and storage resources of cloud servers are deployed at the edge of the radio access network, such as roadside units (RSUs), located near vehicles. This proximity enables service with high QoS and provides a cost-efficient, low-latency solution [6]. VEC offloads vehicle tasks to edge servers located on or near RSUs or to other vehicles with surplus computing resources. Compared to traditional cloud-based systems, edge-based solutions offer significantly lower communication latency by reducing the distance over which tasks are transmitted to computing resources, enhancing responsiveness for latency-sensitive applications [7], [8]. Vehicle

* Manuscript received 2024 September 17, Revised 2024 October 20, Accepted 2024 November 11.

[1] M.Sc. Student, Department of Computer Engineering and IT, University of Kurdistan, Sanandaj, Iran.

[2] Corresponding Author. Associate Professor, Department of Computer Engineering and IT, University of Kurdistan, Sanandaj, Iran.
Email: s.azizi@uok.ac.ir

applications can greatly benefit from the advantages of VEC, leading to a safer and more efficient transportation system [9]. However, optimizing task offloading and resource allocation in VEC remains a fundamental challenge, as these systems must handle heterogeneous user demands, each with unique resource requirements, all while utilizing shared resources. Efficient resource allocation strategies are needed to maximize QoS while minimizing operational costs.

In this paper, we propose a mathematical model for the task offloading and resource allocation problem in heterogeneous VEC systems, formulated as an Integer Linear Programming (ILP) problem. The objective is to maximize the system's total profit while adhering to QoS constraints. We introduce a heuristic weighted algorithm that takes into account the number of tasks in the system and their computational demands. Using this information, the algorithm calculates a weight for each task type and allocates available resources across the MEC servers accordingly. Extensive experiments were conducted across various scenarios to compare the performance of our proposed algorithm against existing algorithms using multiple performance metrics. The results demonstrate that our algorithm significantly improves the system's total profit.

Our main contributions are summarized as follows:

- We present a mathematical framework for addressing task offloading and resource allocation in heterogeneous VEC systems, formulated as an Integer Linear Programming problem, with the goal of optimizing system-wide profit while meeting QoS requirements.
- We propose an innovative heuristic weighted algorithm that dynamically calculates task weights based on both the number of tasks present in the system and their specific computational demands. This algorithm effectively allocates available resources across MEC servers, optimizing system performance by balancing workload and enhancing resource utilization.
- We perform comprehensive experiments across different scenarios to evaluate the performance of the proposed algorithm. These experiments demonstrate that our algorithm consistently outperforms existing approaches across multiple performance metrics, leading to substantial improvements in total system profit, resource efficiency, and overall service quality.

The rest of this paper is organized as follows: In Section 2, we review the related works. Section 3 presents the proposed system model and formulates the optimization problem for task offloading and resource allocation. In Section 4, we introduce our proposed method, a weighted algorithm for task offloading and resource allocation. Section 5 provides an evaluation of our algorithm and simulation results. Finally, Section 6 concludes the paper and discusses future work.

2. RELATED WORK

In recent years, MEC has gained significant attention for its role in computation offloading, leading to the development of various optimization strategies for offloading [10]. Wang et al. [1] introduce a fuzzy logic-based dynamic pricing strategy to optimize offloading decisions and model vehicle interactions as a two-stage Stackelberg game, considering social factors such as reputation and task satisfaction. Wu et al. [11] model the interactions between vehicles and MEC servers using a Markov decision process and optimize decisions using the twin delayed deep deterministic policy gradient (TD3) algorithm. Load balancing is enhanced through edge collaboration and a server selection algorithm based on TOPSIS. Cheng et al. [10] propose the CO-MATCH algorithm, which includes a dynamic programming-based service caching (DPSC) algorithm and a Many-to-One Matching Game (MOMG) algorithm. These components encourage edge services and vehicles to cache tasks and optimize task offloading.

Zhang et al. [12] propose enhancing VECNs with fiber-wireless (FiWi) technology and introduce a software-defined networking (SDN) based load-balancing task offloading scheme. This approach aims to minimize processing delays by efficiently managing computation resources. Fan et al. [7] develop an algorithm using Generalized Benders Decomposition (GBD) and Reformulation Linearization (RL) methods for optimal solutions, as well as a heuristic algorithm for sub-optimal solutions with lower computational complexity. They aim to minimize the total task processing delay by optimizing task scheduling, channel allocation, and computing resource distribution between vehicles and RSUs. Zhao et al. [13] model the joint optimization problem of task offloading and resource allocation as a Markov decision process, taking into account communication, computing, and system costs. They introduce a multi-agent deep deterministic policy gradient (MADDPG) algorithm to address convergence issues in dynamic environments and incorporate federated learning to manage non-IID data and ensure privacy protection.

Du et al. [14] propose a comprehensive IoV architecture and formulate a joint optimization problem to minimize the system function value. They employ a Simulated Spring System Algorithm (SSSA), which decouples the problem into two sub-problems: allocating computing resources based on KKT conditions and optimizing the task offloading strategy using the simulated spring system. These sub-problems iteratively update each other until a solution is achieved. Liu et al. [15] address the challenge of efficient task execution in Vehicular Edge Computing Networks (VECNs) by accounting for the variations in channel and access times due to high vehicle mobility. They propose a multi-path dynamic offloading scheme (MPDOS), designed to minimize the maximum task completion time for vehicles handling serial tasks. MPDOS includes three key components: optimizing communication links to boost processing capability, employing a multi-knapsack algorithm for allocating tasks to RSUs, and implementing a load-balancing scheme to ensure even distribution of computing tasks.

Huang et al. [16] propose a multi-objective optimization model for dynamic, heterogeneous VEC networks, formulated as a multi-objective Markov Decision Process (MOMDP). They introduce EMOTO, a novel multi-objective reinforcement learning algorithm designed to minimize task execution delay and vehicle energy consumption while maximizing service provider revenue. EMOTO integrates a preference priority sampling module and a model-augmented environment estimator to address the challenges of the highly dynamic VEC environment, enhancing decision-making accuracy and efficiency. Wan et al. [17] propose a framework where idle vehicles (IVs) collaborate with busy vehicles (BVs) as edge nodes to reduce task computation latency. They model the matching and resource allocation between BVs and IVs to minimize latency and consider energy consumption, introducing a low-complexity solution for one-to-one matching and an improved biogeography-based optimization (IBBO) algorithm for one-to-many matching. Mao et al. [18] address security challenges in vehicular ad hoc networks (VANETs) by proposing a task offloading mechanism for the IoV that relies on trusted RSUs. They introduce a novel infrastructure trust management model incorporating social factors to enhance RSU security. This mechanism models vehicle task offloading with a focus on RSU reliability, aiming to ensure secure and efficient task processing even under malicious attacks.

Azizi et al. [19] introduce a Mixed-Integer Nonlinear Programming (MINLP) model for task offloading, aimed at maximizing the number of tasks meeting their deadlines while minimizing the overall energy consumption of mobile devices. They propose DECO, a heuristic algorithm designed to optimize the trade-off between task deadlines and energy consumption in IoT devices. DECO jointly considers the deadline requirements of tasks and the energy consumed by devices. Additionally, it accounts for task prioritization and the heterogeneous capabilities of edge cloud servers (ECSs). Yeganeh et al. [20] model task offloading and scheduling in MEC networks as an optimization problem, aiming to minimize execution time and energy consumption. They introduce a hybrid algorithm, E-AEO-AOA, which combines Artificial Ecosystem-based Optimization (AEO) and Arithmetic Optimization Algorithm (AOA). The E-AEO-AOA incorporates a modified Q-learning strategy for hybridization and employs chaos theory to enhance local search capabilities.

While numerous studies have explored resource allocation and task offloading in both MEC and VEC, this work introduces several novel contributions that distinguish it from prior research. First, this study considers a dynamic workload environment, reflecting real-world conditions where task demands fluctuate over time. This added variability increases the complexity of the resource management problem, requiring adaptive strategies to accommodate changing resource requirements effectively. Second, we propose a low-complexity, highly efficient algorithm specifically designed to optimize resource allocation under these fluctuating conditions, ensuring that the computational load is distributed effectively across the available resources. This approach minimizes computational overhead while maintaining responsiveness, an essential factor in edge computing environments. Third, our model incorporates the distinct QoS requirements associated with varying task types. By introducing a specialized objective function, we ensure that resource allocation not only maximizes system performance but also addresses the diverse QoS needs of each task type, which is critical in scenarios where task prioritization and latency sensitivity differ. Through this multi-faceted approach, our work provides a robust framework that balances system performance, resource utilization, and QoS compliance, advancing the state of resource management methodologies in MEC and VEC contexts.

3. System Model

In this section, we provide an overview of our system model and architecture, followed by a detailed discussion on the mathematically modeled problem formulation.

3.1. System Architecture

Fig. 1 illustrates the VEC system model, designed for a two-lane straight road with equally spaced roadside units positioned along the route. Each RSU provides wireless communication coverage, denoted by L. All RSUs are directly connected to a base station which houses a MEC server. Each vehicle is capable of generating multiple tasks, which are categorized into different types. Each task type possesses unique attributes that distinguish it from other task types. Task i generated by a vehicle is characterized by three primary attributes, denoted as $T_i = \{T_i^{type}, T_i^{size}, T_i^{deadline}\}$, where T_i^{type} represents the specific type of the task, T_i^{size} denotes the computational effort required to complete the task, measured in millions of instructions, and $T_i^{deadline}$ indicates the maximum allowable time before the task is rejected, measured in milliseconds for precision.

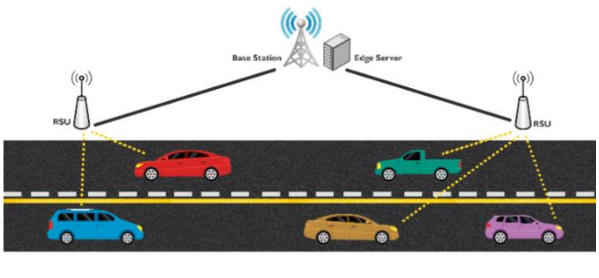


Fig. 1. System architecture

All tasks generated by vehicles are offloaded to the nearest RSU. Upon receiving these tasks, the RSU forwards them to the base station, where the computing process is initiated in the MEC server. **Error! Reference source not found.** illustrates the architecture of the MEC server, which includes three distinct First-In-First-Out (FIFO) queues at the current time (t) for different task types, with each queue linked to a corresponding instance for computation. Each instance is characterized by three

main attributes, denoted as $CI_j = \{CI_j^{state}, CI_j^{cores}, CI_j^{mem}\}$, where CI_j^{state} represents the state of the instance (either on or off), CI_j^{cores} represents the number of CPU cores assigned to the instance, and CI_j^{mem} indicates the memory usage of the instance, measured in megabytes. Each MEC server contains a CPU pool, comprising all the CPU cores available for allocation among the active instances.

Each task type has its own QoS class and constraints to ensure efficient handling and prioritization, enabling flexible resource management by assigning income and penalties based on task completion time to encourage timely task processing and effective system resource management. The details for each class are as follows:

• Task Type 1: This class has a single deadline constraint. If the instance's response time exceeds this standard deadline, the task is rejected, and the system incurs a penalty due to the unmet deadline.

• Task Type 2: The QoS class for Task Type 2 includes two thresholds: the standard deadline and an extended deadline, determined by a multiplier, denoted by θ, of the original deadline. If the task is completed by the standard deadline, full on-time income is earned. Completion after the standard deadline but within the extended θ-adjusted deadline yields reduced income, denoted by β. Exceeding this extended deadline results in task rejection and a penalty.

• Task Type 3: Task Type 3 has three deadline-based QoS thresholds: the standard deadline and two extended deadlines, denoted by $\Delta 1$ and $\Delta 2$. Completion within the standard deadline yields full on-time income. If completed after the standard deadline but before $\Delta 1$, a partial income, denoted by $\Phi 1$, is awarded. Completion between $\Delta 1$ and $\Delta 2$ yields a further reduced income, denoted by $\Phi 2$. Exceeding $\Delta 2$ results in task rejection and a penalty.

3.2. Problem Formulation

In this section, we formulate the task offloading process as an Integer Linear Programming optimization problem. The objective is to maximize the total profit while satisfying constraints such as task deadlines, computational costs, and QoS.

Table 1. Presents symbols and notations used in our problem formulation. We model the number of tasks generated for each type within a time slot using a Poisson distribution, where each task type has a unique generation rate, denoted by λ_m for task type *m*. Upon the task's entry into the MEC server, the server calculates the response time for the task. The response time for task *i* on container *j* is expressed as:

$$T_{i,j}^{res} = Q_j^{wait} + \frac{T_i^{size}}{(CI_j^{cores} * CP)} \tag{1}$$

where CP represents the CPU power of each core, measured in million instructions per second (MIPS), and is assumed to be homogeneous across the system. Since a FIFO system has been implemented for each queue, Q_j^{wait} represents the waiting time for task execution in the queue of container *j* and is expressed as:

$$Q_j^{wait} = \frac{\sum_{k=1}^{N_j} T_k^{size}}{(CI_j^{cores} * CP)} \tag{2}$$

where N_j represents the total number of tasks waiting to be executed in the queue of container *j* before task i.

Once the response time is calculated, the system determines whether the server can execute task *i* before its deadline, denoted as $T_i^{deadline}$. Based on the task type and the QoS class, the system decides whether to reject the task or add it to the instance's queue for execution. If the task is accepted, the associated income is added to the total income of the container, denoted by CI_j^{inc}. Conversely, if the task is rejected, the penalty incurred by the system is added to the total penalty of the container, denoted by CI_j^{pen}. Therefore, the income for each task type can be expressed as:

$$CI_1^{inc} = OT_1 \times TT_1^{otp} \tag{3}$$

and

$$CI_2^{inc} = (OT_2 \times TT_2^{otp}) + (TH \times \beta) \tag{4}$$

and

$$CI_3^{inc} = (OT_3 \times TT_3^{otp}) + (D1T \times \Phi 1) + (D2T \times \Phi 2) \tag{5}$$

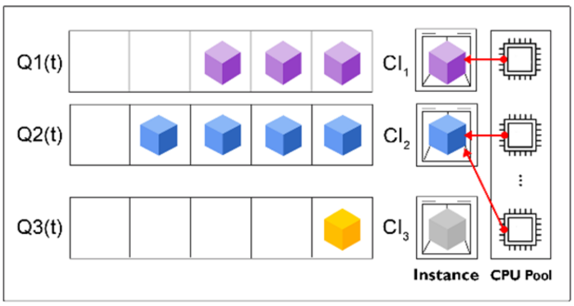


Fig2 MBC server model

TABLE 1
Key notations used in the problem formulation

Symbol	Description	Unit
λ_m	Task generation rate for task type *m*	-
T_i^{size}	Size of task *i*	[MI]
$T_i^{deadline}$	Deadline for task *i*	[ms]
OT_m	Number of tasks executed before the original deadline for task type m	-
TT_m^{otp}	Income earned for completing a task on time for task type *m*	[$]
TH	Number of tasks executed within the time frame $\theta \times T_i^{deadline}$ but after the original deadline	-
β	Income earned for completing tasks within the time frame $\theta \times T_i^{deadline}$	[$]
$D1T$	Number of tasks completed between the original deadline and $\Delta 1 \times T_i^{deadline}$	-
Φ1	Profit for completing a task during the time frame $\Delta 1 \times T_i^{deadline}$	[$]
$D2T$	Number of tasks executed between $\Delta 1 \times T_i^{deadline}$ and $\Delta 2 \times T_i^{deadline}$	-
Φ2	Profit for completing tasks within the time frame $\Delta 2 \times T_i^{deadline}$	[$]
RJT_m	Number of rejected tasks for task type *m*	-
TT_m^{pen}	Penalty amount for rejected tasks for task type *m*	[$]
TT_m^{mem}	Memory usage for task type *m*	[$]
TT_m^{mc}	Cost per 100 MB of memory for task type *m*	[$]
CI_j^{cores}	Number of CPU cores allocated to container *j*	-
CP	CPU power of each core	[MIPS]
Q_j^{wait}	Waiting time for task execution in the queue of container	[ms]
N_j	Total number of tasks waiting to be executed in the queue of container *j* before task i	-
CI_j^{inc}	Total income earned by container *j*	[$]
CI_j^{pen}	Total penalty incurred by container *j* for rejected tasks	[$]
RT_j^{idle}	Idle runtime for container *j*	[ms]
RT_j^{active}	Active runtime for container *j*	[ms]
CI_j^{mem}	Memory usage of container *j*	[MB]
$CI_j^{totalmem}$	Total memory usage for container *j*	[MB]
$CI_j^{memcost}$	Total memory cost for container *j*	[$]
$PROF_j$	Profit of container *j*	[$]
$T_{i,j}^{res}$	Response time for task *i* on container *j*	[ms]
IN^{total}	Total income earned across all containers	[$]
PEN^{total}	Total penalty incurred across all containers	[$]
$PROF^{total}$	Total profit of the system	[$]

where, for all task types, OT_m represents the number of tasks executed before the original deadline, TT_m^{otp} is the amount of income earned for executing a task on time. For task type 2, TH denotes the number of tasks executed before $\theta \times T_i^{deadline}$ and after the original deadline, with β being the income earned for executing tasks within this timeframe. For task type 3, $D1T$ represents the number of tasks completed after the original deadline but before $\Delta 1 \times T_i^{deadline}$, with Φ1 indicating the profit for executing a task during this period. Similarly, $D2T$ is the number of tasks executed after $\Delta 1 \times T_i^{deadline}$ and before $\Delta 2 \times T_i^{deadline}$, with Φ2 being the profit for completing tasks within this timeframe. Therefore, the total income can be expressed as:

$$IN^{total} = \sum_{j=1}^{3} CI_j^{inc} \quad (6)$$

The penalty is determined by the number of rejected tasks, with each task type incurring a different penalty amount. Thus, the total penalty amount for each task type can be expressed as:

$$CI_m^{pen} = RJT_m \times TT_m^{pen} \quad (7)$$

where RJT_m represents the number of rejected tasks for task type *m*, and TT_m^{pen} is the penalty amount assigned to that task type. The total penalty for all containers can be expressed as:

$$PEN^{total} = \sum_{j=1}^{3} CI_j^{pen} \quad (8)$$

In this system, we consider two types of runtimes for each container. The first is idle runtime, which occurs when the container's state is 'On' but no tasks are being executed. This is denoted by RT_j^{idle}. The second type is active runtime, which occurs when the container's state is 'On' and a task is being executed. This is denoted by RT_j^{active}. As previously mentioned, each container has its own memory usage, denoted by CI_j^{mem}. Additionally, each task type has its own memory usage, which can be expressed as TT_m^{mem}

for task type *m*. Therefore, the total memory usage for container *j* and task type *m* can be expressed as:

$$CI_j^{totalmem} = \left(RT_j^{active} \times TT_m^{mem}\right) + \left(RT_j^{idle} \times CI_j^{mem}\right) \tag{9}$$

and the total memory cost for container *j* and task type *m* can be expressed as:

$$CI_j^{memcost} = CI_j^{totalmem} \times \frac{TT_m^{mc}}{100} \tag{10}$$

where TT_m^{mc} represents the cost per 100 megabytes of memory for task type *m*. For each container, our goal is to maximize its profit, so the profit for container *j* can be expressed as:

$$PROF_j = CI_j^{inc} - CI_j^{pen} - CI_j^{memcost} \tag{11}$$

and the total profit of the system can be expressed as:

$$PROF^{total} = \sum_{j=1}^{3} PROF_j \tag{12}$$

To reiterate, our optimization objective is to maximize the total profit of the system; therefore, the total profit serves as our objective function. In the following section, we propose a weighted algorithm to solve this optimization problem. This algorithm is designed to allocate tasks to the appropriate instances efficiently while considering multiple factors such as task deadlines, resource availability, and QoS.

4. PROPOSED ALGORITHM

In this section, we propose a heuristic weighted task offloading and resource allocation algorithm. To design this weighted algorithm, we make the following assumptions:

Assumption 1: We assume that the task generation for each task type follows a Poisson distribution, with a given generation rate λ and an average task size, denoted as $TT^{avgsize}$.

Assumption 2: We assume the size of each task follows a normal distribution, with $TT^{avgsize}$ as the mean and a standard deviation of σ.

Assumption 3: The number of epochs within a specific timeframe is assumed to be known.

Assumption 4: All instances are assumed to remain in the '*On*' state during each epoch and are allocated at least one CPU core.

As mentioned previously, our system design is based on time slots. The total number of time slots is divided into a series of epochs, during which the task offloading and resource allocation algorithm is executed. For each epoch, the task generation rate varies across all three task types. At the beginning of each epoch, our algorithm makes two key decisions: first, it determines which containers will be in the 'Off' state and which will remain '*On*'; second, it decides the number of CPU cores to allocate from the CPU pool to each container that is in the 'On' state. As stated in our assumptions, we assume that all instances remain in the '*On*' state, and the number of CPU cores allocated to each instance is determined by the weight of the queue for each task type. The weight calculation for each task type can be expressed as:

$$w_m^{ep} = \lambda_m^{ep} \times TT_m^{avgsize} \tag{13}$$

where λ_m^{ep} represents the generation rate and $TT_m^{avgsize}$ represents the average task size of task type *m* in epoch number *ep*. The total weight of all task types can be expressed as:

$$w_{total}^{ep} = \sum_{j=1}^{3} w_j^{ep} \tag{14}$$

We use the weight for each task type to calculate a coefficient, which is then utilized to assign a specific number of CPU cores to each task type's instance in the current epoch. The coefficient for task type *m* in epoch *ep* can be expressed as:

$$co_m^{ep} = \frac{w_m^{ep}}{w_{total}^{ep}} \tag{15}$$

The number of CPU cores assigned to the instance of task type *m*, denoted by CI_m^{cores}, can be specified as:

$$CI_m^{cores} = \left\lfloor co_m^{ep} \times AC \right\rfloor \tag{16}$$

where *AC* represents the total number of available CPU cores in the CPU pool. After assigning the floor of the calculated number of CPU cores based on the coefficient to each instance, there may be some remaining extra cores that need to be allocated. For each instance, a "luck percentage," denoted by CI_m^{luck} is assigned, with the sum of all luck percentages totaling 100%. These extra cores are distributed to instances based on their luck percentage. For example, if $CI_m^{luck} = 70$, it means there is a 70% chance that an extra core will be allocated to that instance. For each extra core, a random number between 0 and 100 is generated, and if the number falls within the range of CI_m^{luck}, the extra core is added to CI_m^{cores}. The pseudocode of the proposed weighted algorithm is presented in Algorithm 1.

As presented in the pseudocode, the algorithm iterates through each epoch, with the maximum number of epochs denoted as ec, allowing it to adapt dynamically to changing conditions. Lines 2 through 5 calculate the weight for each task type, with line 5 computing the total weight across all task types to provide an overview of the computational demands during the current epoch. Line 6 then initiates an iteration through each instance, with lines 7 and 8 ensuring that each instance is allocated at least one CPU core to maintain operational integrity. Using the total weight, lines 9 through 12 calculate a coefficient for each task type, determining the proportion of resources allocated based on their weighted importance. The algorithm then allocates the floor of the calculated number of cores to each instance to ensure feasibility. Lastly, lines 13 through 19 address the distribution of any remaining cores, which are allocated individually based on each instance's luck percentage if extra cores remain after the initial allocation.

In analyzing the time complexity of the proposed algorithm, the outer loop iterates ec times, resulting in a time complexity of $O(ec)$. The two inner loops in lines 2 through 4 and lines 6 through 12 each run a constant

number of times (specifically, three iterations), giving them a time complexity of $O(1)$. Therefore, these inner loops do not impact the overall complexity. Lines 13 through 19 consist of a while loop that executes up to AC times, contributing a time complexity of $O(AC)$. Consequently, combining the contributions from the outer loop and the while loop, the total complexity of the proposed algorithm is $O(ec \times AC)$.

Algorithm 1 Proposed Resource Allocation Algorithm

Input: $AC, \lambda_m^{ep}, TT_m^{avgsize}, ec$
Output: CI_m^{cores}
1: **for** *ep=0;ep<ec;ep++* **do**
2:     **for** *m=0;m<3;m++* **do**
3:         calculate weight for each task type *m* in epoch *ep:* $w_m^{ep} = \lambda_m^{ep} \times TT_m^{avgsize}$
4:     **end for**
5:     calculate total weight for the current epoch:

$$w_{total}^{ep} = \sum_{m=1}^{3} w_j^{ep}$$

6:     **for** *m=0;m<3;m++* **do**
7:         $CI_m^{cores} = CI_m^{cores} + 1$
8:         $AC = AC - 1$
9:         calculate coefficient for task type *m* in epoch *ep:* $co_m^{ep} = \frac{w_m^{ep}}{w_{total}^{ep}}$
10:        allocate the floor of calculated number of cores to the instance of task type *m:* $CI_m^{cores} = \lfloor co_m^{ep} \times AC \rfloor$
11:        $AC = AC - CI_m^{cores}$
12:    **end for**
13:    **while** $AC > 0$ do
14:        generate a random number from 0 to 100
15:        **if** Random number is in range of CI_m^{luck} **do**
16:            $AC = AC - 1$
17:            $CI_m^{cores} = CI_m^{cores} + 1$
18:        **end if**
19:    **end while**
20: **end for**
21: **return** CI_m^{cores}

5. EVALUATION

In this section, we evaluate the performance of our proposed weighted algorithm by conducting a series of simulation experiments under various scenarios and comparing the corresponding numerical results.

5.1. Simulation Setup

Our experiment was conducted using Python 3.10.4 to simulate a VEC environment with multiple users. The simulation was performed on a computer with the following specifications: an AMD Ryzen 7 5800X3D processor, 32 GB of RAM, and an NVIDIA RTX 3080 GPU. The total number of time slots was set to 1000, divided into four epochs. Each time slot had a duration of 1000 ms, with the time slot indices assigned to the epochs being [0, 250, 500, 750]. TABLE 2 presents other simulation parameters used in the experiments.

TABLE 2
Simulation parameters

Parameter	Value
Number of CPU cores	15
CPU core power	2000 MIPS
Set of task types	[1, 2, 3]
Task type deadlines	[200, 800, 4000] (ms)
Task type rejection penalties	[-0.2, -0.05, -0.02] ($)
Task type size means	[200, 1000, 4000]
Task type size standard deviations	[20, 100, 400]
On time income	[0.5, 0.8, 1] ($)
β income for task type 2	0.5 $
Φ1 and Φ2 income for task type 3	[2, 4] ($)
Memory usage for task types	[0.1, 0.2, 0.5] (MB/ms)
θ	1.5
Δ1, Δ2	[2, 4]
Cost of memory per 100 MB	[0.002, 0.004, 0.008] ($)
Idle memory usage for instances	[200, 500, 1000] (MB/ms)

5.2. Experiment Scenarios

We compared our proposed weighted algorithm with other algorithms across three distinct scenarios, each characterized by unique task generation rates for every task type within each epoch. This dynamic task generation rate allowed us to assess the effectiveness of our algorithm and its impact on the metrics considered in this experiment. The task generation parameters for all three scenarios are presented in TABLE 3. For each task type in each scenario, a list is provided, where the index corresponds to the epoch index, and the value represents the task generation rate for that task type in the respective epoch.

TABLE 3
Scenarios parameters

	λ_1	λ_2	λ_3
Scenario 1	[4, 16, 10, 2]	[4, 6, 6, 0]	[2, 3, 4, 1]
Scenario 2	[10, 2, 13, 0]	[2, 1, 0, 4]	[4, 1, 3, 2]
Scenario 3	[1, 8, 5, 12]	[0, 4, 4, 6]	[3, 3, 1, 4]

5.3. Compared Algorithms

We evaluated the performance of our proposed weighted algorithm by comparing it with the following algorithms:
1) **RAND-RAND**: In this approach, the state of each instance is assigned randomly (either On or Off), and the

number of cores allocated to each instance in the 'On' state is also determined randomly.

2) **WARM-EVEN** [21]: This algorithm maintains all instances in a warm state (On) and distributes the CPU cores evenly among them.

3) **WARM-RAND** [22]: In this method, all instances are kept in a warm state, and the allocation of CPU cores is performed randomly.

5.4. Metrics

In this subsection, we present the metrics used to evaluate the performance of our proposed algorithm. These metrics offer a comprehensive assessment of the algorithm's effectiveness in comparison to other algorithms. The selected metrics are designed to capture various dimensions of performance and include:

1) Memory Cost: This metric quantifies the total memory cost, calculated based on memory consumption across all instances in each scenario. It considers both active and idle runtime, as outlined in Eq. (9) and Eq. (10).

2) Penalty: This metric assesses the total penalty incurred by the system for each algorithm within each scenario. The penalty is determined by the number of rejected tasks, reflecting the algorithm's impact on task acceptance

3) Income: This metric measures the total net income earned by the system in each scenario. It captures the financial performance of the system based on task execution and resource utilization.

4) Profit: This metric calculates the total profit by accounting for both costs and income, as described in Eq. (12). It provides an overarching measure of the algorithm’s effectiveness in optimizing the system’s financial outcomes.

These metrics collectively provide a comprehensive view of the algorithm's performance, enabling a detailed comparison with other algorithms.

5.5. Results

In this subsection, we analyze the different metrics in our experiments and review the numeral results. As shown in Fig. 2, the RAND-RAND algorithm exhibits lower memory costs, primarily because instances can be completely turned off during some epochs, thereby reducing memory consumption. However, this approach has several drawbacks, including a decrease in QoS due to an increase in rejected tasks. This increase in rejected tasks leads to higher penalties incurred by the system and ultimately results in a reduction in total profit. Examining the numerical results of the other three algorithms, where all instances remain in the 'On' state, we observe that our proposed algorithm reduces memory cost by an average of 2.54% across all three scenarios compared to the WARM-EVEN algorithm. Additionally, WARM-EVEN outperforms WARM-RAND by an average of 2.17%.

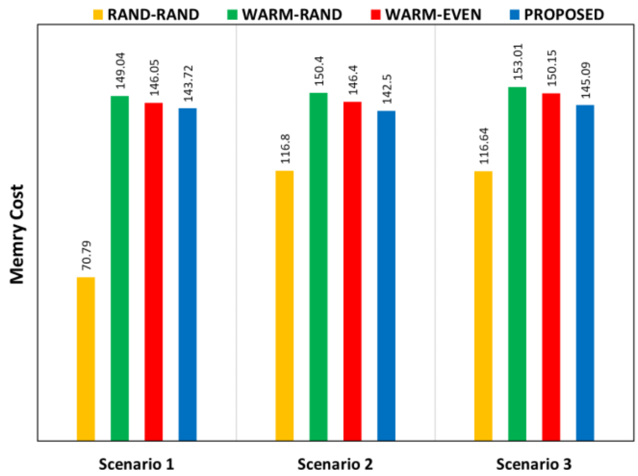


Fig. 2. Comparison of total memory cost

Fig. 3 illustrates the impact of penalties incurred by the system as a result of the number of rejected tasks. Our proposed algorithm demonstrates superior performance compared to the other three algorithms, achieving an average penalty reduction of 11.4% relative to the WARM-EVEN algorithm, which outperforms the other two algorithms in this metric. Notably, in scenario two, our proposed algorithm achieves a remarkable 25.98% reduction in penalties. This significant improvement is attributed to the algorithm's sophisticated allocation strategy, which intelligently assigns CPU cores to each instance based on the load of each task type's queue. By effectively managing resources and minimizing task rejection, our algorithm reduces the associated penalties and enhances overall system efficiency. This reduction in penalties highlights the algorithm's ability to optimize resource use and improve system performance, even when task loads and demands fluctuate.

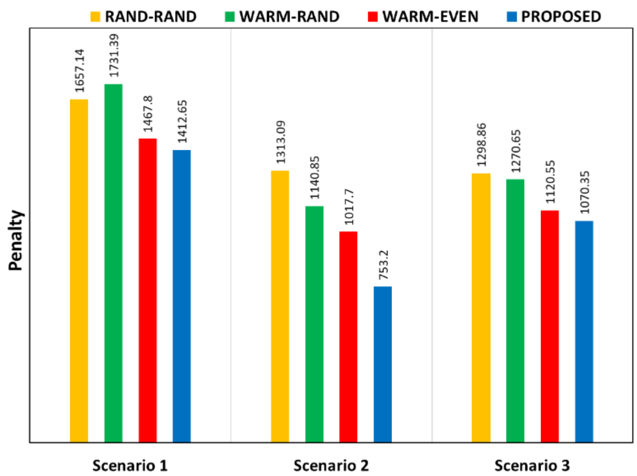


Fig. 3. Comparison of total penalty

Fig. 4 illustrates the impact of different algorithms on system income, highlighting that our proposed weighted

algorithm consistently demonstrates superior optimization performance. It can be observed that our proposed algorithm outperforms the other three algorithms across all three scenarios. On average, our proposed algorithm achieves a 10.43% increase in income compared to the WARM-EVEN algorithm, which consistently outperforms the remaining two algorithms across all scenarios in this metric. In particular, scenario 2 reveals an even more pronounced advantage, with our algorithm exceeding the income earned by WARM-EVEN by 18.79%. This enhancement in performance is primarily due to the proposed algorithm's more effective resource allocation strategy. By optimizing the distribution of resources, the algorithm enables a higher number of tasks to be executed. Since each successfully executed task contributes to the overall income of the system, this improved allocation significantly boosts the system's total income. The results represent the impact of our proposed algorithm in maximizing system income through better management of computational resources.

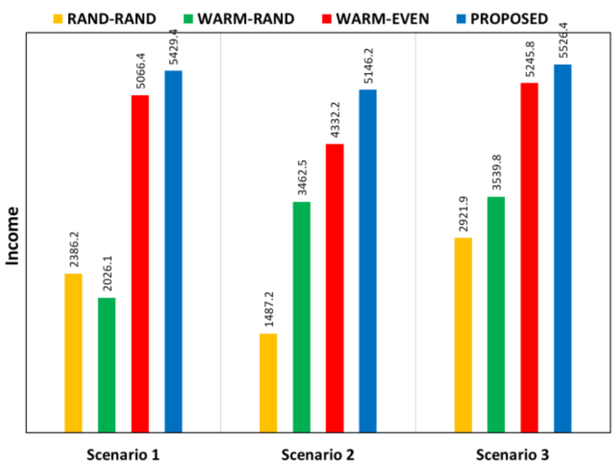


Fig. 4. Comparison of total income

Fig. 5 represents the total profit of the system which is the main objective of this optimization problem. When comparing the algorithms across all three scenarios, it is evident that our algorithm outperforms the other three by a significant margin. On average, the weighted algorithm increases the system's total profit by 18.26% compared to the WARM-EVEN algorithm, which has the best performance among the remaining three. The most notable improvement is observed in scenario 2, where the total profit increases by 34.16%. This substantial gain is attributable to the weighted algorithm's capacity to minimize system penalties by maintaining instances in a warm state, thereby reducing memory consumption and overall costs. Moreover, the algorithm facilitates the execution of a greater number of tasks through efficient resource allocation, tailored to the load of each instance. This enhanced resource management not only boosts the total income but also contributes to a considerable increase in the system's total profit. The synergy of effective instance management and optimized resource allocation results in a markedly improved overall system profit.

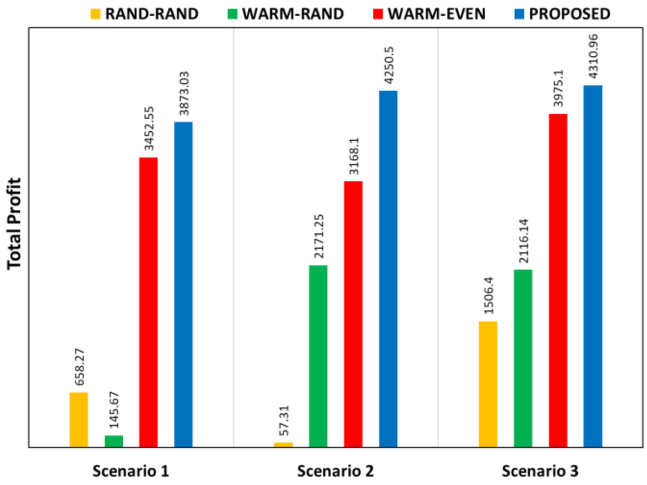


Fig. 5. Comparison of total profit

6. Conclusion and Future Work

In this paper, we reviewed the task offloading and resource allocation problem in vehicular edge computing and formulated the problem mathematically. We proposed a weighted algorithm to optimize this problem, considering QoS, cost, and profit. We compared our algorithm across three different scenarios with other algorithms based on metrics such as memory cost, penalties, income, and profit. The results showed that our algorithm increased total profit by an average of 18.26%, while also reducing total costs and increasing system income. These experiments demonstrate that our algorithm is well-suited for real-time environments due to its low response time. This study did not account for heterogeneous CPU cores, which could be explored in future research. Additionally, energy consumption should be addressed by implementing strategies such as turning off unused instances, disabling unnecessary cores, or leveraging technologies like Dynamic Voltage and Frequency Scaling (DVFS) to reduce energy usage. Furthermore, utilizing deep learning methods to predict system workload may enhance resource allocation and overall responsiveness. These aspects could be incorporated into future work to further improve the proposed solution.

References

[1] S. Wang, D. He, M. Yang, and L. Duo. (2024, Oct.). Cost-aware task offloading in vehicular edge computing: A Stackelberg game approach. *Vehicular Communications*. [Online]. 49(202001), p. 100807. Available: https://doi.org/10.1016/j.vehcom.2024.100807

[2] L. L. Wang, J. S. Gui, X. H. Deng, F. Zeng, and Z. F. Kuang. (2020, Dec.). Routing Algorithm Based on Vehicle Position Analysis for Internet of Vehicles. *IEEE Internet of Things Journal*. [Online]. 7(12), pp. 11701–11712. Available: https://doi.org/10.1109/JIOT.2020.2999469

[3] M. K. Farimani, S. Karimian-Aliabadi, R. Entezari-

Maleki, B. Egger, and L. Sousa. (2024, Sep.). Deadline-aware task offloading in vehicular networks using deep reinforcement learning. *Expert Systems with Applications.* [Online]. 249(PB), p. 123622. Available: https://doi.org/10.1016/j.eswa.2024.123622

[4] P. Mach and Z. Becvar. (2017, Mar.). Mobile Edge Computing: A Survey on Architecture and Computation Offloading. *IEEE communications surveys & tutorials* [Online]. 19(3), pp. 1628–1656. Available: https://doi.org/10.1109/COMST.2017.2682318

[5] L. Liu, C. Chen, Q. Pei, S. Maharjan, and Y. Zhang. (2021, Jun.). Vehicular Edge Computing and Networking: A Survey. *Mobile networks and applications.* [Online]. 26(3), pp. 1145–1168. Available: https://doi.org/10.1007/s11036-020-01624-1

[6] M. Khayyat, I. A. Elgendy, A. Muthanna, A. S. Alshahrani, S. Alharbi, and A. Koucheryavy. (2020, Jul.). Advanced Deep Learning-Based Computational Offloading for Multilevel Vehicular Edge-Cloud Computing Networks. *IEEE Access*. [Online]. 8, pp. 137052–137062. Available: https://doi.org/10.1109/ACCESS.2020.3011705

[7] W. Fan et al. (2023, Jan.). Joint Task Offloading and Resource Allocation for Vehicular Edge Computing Based on V2I and V2V Modes. *IEEE Transactions on Intelligent Transportation Systems.* [Online]. 24(4), pp. 4277–4292. Available: https://doi.org/10.1109/TITS.2022.3230430

[8] R. Salimi, S. Azizi, and J. Abawajy. (2024, May.). A greedy randomized adaptive search procedure for scheduling IoT tasks in virtualized fog–cloud computing. *Transactions on Emerging Telecommunications Technologies*. [Online]. 35(5). Available: https://doi.org/10.1002/ett.4980

[9] F. Gu, X. Yang, X. Li, and H. Deng. (2022, Jul.). Computational Resources Allocation and Vehicular Application Offloading in VEC Networks. *Electronics*. [Online]. 11(14), pp. 1–16. Available: https://doi.org/10.3390/electronics11142130

[10] C. Cheng, L. Zhai, X. Zhu, Y. Jia, and Y. Li. (2024, Aug.). Dynamic task offloading and service caching based on game theory in vehicular edge computing networks. *Computer Communications*. [Online]. 224, pp. 29–41. Available: https://doi.org/10.1016/j.comcom.2024.05.020

[11] Z. Wu, Z. Jia, X. Pang, and S. Zhao. (2024, Apr.). Deep Reinforcement Learning-Based Task Offloading and Load Balancing for Vehicular Edge Computing. *Electronics*. [Online]. 13(8). Available: https://doi.org/10.3390/electronics13081511

[12] J. Zhang, H. Guo, J. Liu, and Y. Zhang. (2019, Dec.). Task Offloading in Vehicular Edge Computing Networks: A Load-Balancing Solution. *IEEE Transactions on Vehicular Technology*. [Online]. 69(2), pp. 2092–2104. Available: https://doi.org/10.1109/TVT.2019.2959410

[13] X. Zhao, Y. Wu, T. Zhao, F. Wang, and M. Li. (2024, Sep.). Federated deep reinforcement learning for task offloading and resource allocation in mobile edge computing-assisted vehicular networks. *Journal of Network and Computer Applications*. [Online]. 229, p. 103941. Available: https://doi.org/10.1016/j.jnca.2024.103941

[14] Z. Du, Y. Ni, H. Tao, and M. Yin. (2024, Nov.). Joint optimization of offloading strategy and resource allocation for multi-user in dynamic vehicular edge computing systems. *Simulation Modelling Practice and Theory*. [Online]. 136, p. 103001. Available: https://doi.org/10.1016/j.simpat.2024.103001

[15] X. Liu, J. Zheng, Y. Li, M. Zhang, R. Wang, and Y. He. (2024, Oct.). Multi-path serial tasks offloading strategy and dynamic scheduling optimization in vehicular edge computing networks. *Vehicular Communications*. [Online]. 49, p. 100827. Available: https://doi.org/10.1016/j.vehcom.2024.100827

[16] Z. D. Huang, X. F. Wu, and S. Bin Dong. (2024, Sep.). Multi-objective task offloading for highly dynamic heterogeneous Vehicular Edge Computing: An efficient reinforcement learning approach. *Computer Communications*. [Online]. 225, pp. 27–43. Available: https://doi.org/10.1016/j.comcom.2024.06.018

[17] N. Wan, Y. Luo, G. Zeng, and X. Zhou. (2022, Sep.). Minimization of VANET execution time based on joint task offloading and resource allocation. *Peer-to-Peer Networking and Applications*. [Online]. 16(1), pp. 71–86. Available: https://doi.org/10.1007/s12083-022-01385-6

[18] M. Mao, T. Hu, and W. Zhao. (2023, Feb.). Reliable task offloading mechanism based on trusted roadside unit service for internet of vehicles. *Ad Hoc Networks*. [Online]. 139, p. 103045. Available: https://doi.org/10.1016/j.adhoc.2022.103045

[19] S. Azizi, M. Othman, and H. Khamfroush. (2022, Jul.). DECO: A Deadline-Aware and Energy-Efficient Algorithm for Task Offloading in Mobile Edge Computing. *IEEE Systems Journal*. [Online]. 17(1), pp. 952–963. Available: https://doi.org/10.1109/JSYST.2022.3185011

[20] S. Yeganeh, A. Babazadeh Sangar, and S. Azizi. (2023, May.). A novel Q-learning-based hybrid algorithm for the optimal offloading and scheduling in mobile edge computing environments. *Journal of Network and Computer Applications*. [Online]. 214, p. 103617. Available: https://doi.org/10.1016/j.jnca.2023.103617

[21] A. Fuerst and P. Sharma. (2021, Apr.). FaasCache: Keeping serverless computing alive with greedy-dual caching. *Proceedings of the 26th ACM international conference on architectural support for programming languages and operating systems*. [Online]. pp. 386–400. Available: https://doi.org/10.1145/3445814.3446757

[22] H. Ko and S. Pack. (2022, Sep.). Function-Aware Resource Management Framework for Serverless Edge Computing. *IEEE Internet of Things Journal*. [Online]. 10(2), pp. 1–10. Available: https://doi.org/10.1109/JIOT.2022.3205166